Mining Parametric Specifications

Feng Chen
Department of Computer Science
University of Illinois at Urbana-Champaign
fengchen@cs.uiuc.edu

Choonghwan Lee
Department of Computer Science
University of Illinois at Urbana-Champaign
clee83@cs.uiuc.edu

Grigore Roşu
Department of Computer Science
University of Illinois at Urbana-Champaign
grosu@cs.uiuc.edu

ABSTRACT

Mining formal specifications from program executions has numerous applications in software analysis, from program understanding and modeling to testing and bug detection.

Parametric specifications carry parameters that are bound to concrete values at runtime. They are useful for specifying system behaviors involving multiple components. Runtime monitoring of parametric specifications is relatively well-understood, with several performant runtime monitoring systems available. The main challenge underlying such parametric monitoring systems is to slice parametric execution traces into smaller, non-parametric traces, each relevant for a particular parameter instance; then each of the trace slices is monitored against a non-parametric monitor.

This paper presents a novel technique to automatically mine parametric specifications from execution traces, which builds upon the observation that there is an inherent duality between parametric specification monitoring and parametric specification mining: they both rely on an online parametric trace slicing process, followed either by monitoring the resulting trace slices against given specifications in the first case, or by inferring the specifications that best explain the observed trace slices in the second case.

A blind use of off-the-shelf parametric trace slicing techniques from monitoring leads not only to inefficient but also to "noisy" slicers for mining; "noise" in this context means trace slices that are irrelevant for the desired property, so they perturb the learning process. Our first contribution is a mining-specific parametric trace slicing algorithm, which makes slicing not only feasible, but also precise for mining. The obtained trace slices can then be passed to any non-parametric property learner. A blind use of the off-the-shelf Probabilistic Finite State Automata (PFSA) learner leads to over-generalized specifications in many experiments. Thus, our second contribution is a refinement of the PFSA learner that results in accurate mining of specifications.

The presented technique has been implemented in a prototype tool for Java. We have applied it to a number of real-world programs, including Java class libraries and popular open source packages. Our experiments generated many meaningful specifications, and revealed tricky bugs in some open source programs.

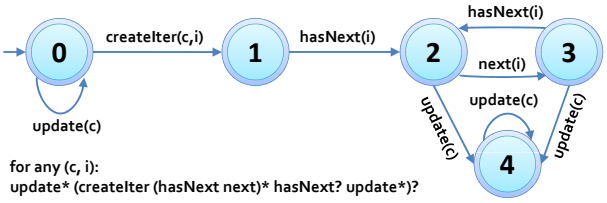

Figure 1: Mined Collection–Iterator usage pattern

1. INTRODUCTION

Automatically mining specifications, e.g., API patterns and usage scenarios, from observed executions is receiving increasing research interest due to its effectiveness in improving software development, e.g., allowing for better program understanding, facilitating bug detection and program verification, and smoothing software evolution. Numerous approaches [2, 17, 28, 13, 1, 21] have been proposed, aiming at mining different kinds of specifications. However, none of them gives a satisfactory solution for parametric specifications in their full generality. Parametric specifications carry free variables, called parameters, which are instantiated to concrete values at runtime. They provide a natural and effective way to specify properties involving one or more components, e.g., interaction protocols among different objects in object-oriented programming.

Figure 1 illustrates a usage pattern involving two Java classes, Collection and Iterator, automatically[1] mined by our tool. The pattern is specified as a deterministic finite automaton (DFA) and an equivalent regular expression. Collection is the base class for implementing collection-type data structures, e.g., lists and sets. Iterator is an interface used to iterate through a Collection. In Figure 1, update is the event representing a method call on a Collection object that changes the contents of the collection, such as add or remove. It has the target Collection object as parameter. createIter is the event used to create an Iterator object from a Collection object and has two parameters: the underlying Collection object and the created Iterator object. hasNext and next represent invocations on methods hasNext and next of Iterator, respectively. The former checks whether the iterator has more elements and the latter fetches the next element.

The specification in Figure 1 states the following safety

[1]The only input we require from user is the event definitions.

```
...
 1. update(Collection:158)
 2. createIter(Collection:158, Iterator:119)
 3. hasNext(Iterator:119)
 4. next(Iterator:119)
 5. update(Collection:148)
 6. hasNext(Iterator:119)
 7. next(Iterator:119)
 8. update(Collection:148)
 9. hasNext(Iterator:119)
10. next(Iterator:119)
11. update(Collection:148)
12. hasNext(Iterator:119)
13. createIter(Collection:263, Iterator:131)
14. hasNext(Iterator:131)
...
```

Figure 2: Fragment of a logged trace

property: if an iterator i is created for a collection c, the contents of c should not be changed while i is being used; indeed, once the automaton enters state 4, no method calls of the iterator are allowed (state 4 does not accept hasNext or next). A violation of this property will cause a runtime exception in Java. Figure 1 also shows another usage pattern of Iterator: every call to the next method should be guarded by a hasNext method call. This pattern is not mandatory, but it indicates the desired behavior of an Iterator; its violations may imply unsafe uses of Iterator (a call to the next method of an iterator that has no elements will cause a runtime exception, which can be hard to reproduce and debug). In fact, we found a violation of this pattern in the pmd benchmark in the DaCapo [5] suite, which turned out to be a subtle bug (see Section 6).

Despite their usefulness in practice, parametric specifications are much more challenging to mine than non-parametric specifications, mainly because of the complexity of handling parameter information contained in the observed executions. Figure 2 shows a fragment of an execution trace obtained in our experiments to infer the specification in Figure 1. This trace is a *parametric trace*, i.e., a trace containing events with parameter bindings. For example, the first event in the trace, update(Collection:158), instantiates the parameter Collection with a concrete object represented using the first three digits of its runtime hashcode[2], 158. Note that there can be other ways to encode parametric traces, such as the one in [2], but they are essentially equivalent. As a parametric trace is usually comprised of many meaningful traces merged, each for a particular parameter binding, accurately extracting each meaningful trace is necessary. For example, in Figure 2, the trace for binding (Iterator:131) is hasNext (the last event) and the trace for (Iterator:119) is hasNext next hasNext next hasNext next hasNext (events 3, 4, 6, 7, 9, 10 and 12). If these two traces are not treated separately, a miner would likely infer a specification that contains two consecutive hasNexts. Additionally, these traces can interleave and sometimes overlap, imposing a non-trivial challenge to correctly separating them. Moreover, the number of parameter bindings can be tremendous. In many of our experiments, we observed more than a million instantiations of the given parameters, making it highly difficult to handle the parameter information efficiently.

Most existing approaches handle parameters in a limited and often implicit way to avoid the complexity of parameter handling, resulting in restricted capability of mining parametric specifications. For example, *none of the approaches* that we are aware of are able to infer the specification in Figure 1, as discussed in Section 2. In this paper, we propose an effective and generic approach to mine parametric specifications. First, we introduce an efficient algorithm to slice an observed parametric trace into a set of non-parametric trace slices (i.e., strings of events), each corresponding to a particular parameter binding. Figure 3 shows the resulting set of non-parametric trace slices with their corresponding parameter bindings for Figure 2. The computed set of non-parametric trace slices is fed into a specification learner to infer a non-parametric specification. Also, we extend the PFSA [23] learning algorithm to mine compact and accurate state-based specifications. The generated non-parametric specification is then associated with parameters to obtain the final parametric specification. This way, we separate the tasks of parameter handling and of specification learning. This separation gives us the following advantages: 1) the parameter handling process makes no assumption on the types of specifications to mine, resulting in a generic mining framework for parametric specifications; 2) the learning algorithm is not required to handle parameter information, significantly reducing the effort to re-use an existing algorithm or to develop a new one. In summary, the main contributions of this paper include:

```
(Collection:158)              : update
(Collection:158, Iterator:119): update createIter hasNext
                    next hasNext next hasNext next hasNext
(Iterator:119)                : hasNext next hasNext next
                                hasNext next hasNext
(Collection:148)              : update update update
(Collection:263, Iterator:131): createIter hasNext
(Iterator:131)                : hasNext
```

Figure 3: Non-parametric trace slices contained in Figure 2

1. An efficient algorithm to slice a parametric trace into a set of non-parametric trace slices according to a given set of parameters; this algorithm provides a generic framework for mining parametric specifications.

2. An extended PFSA learning algorithm that infers compact and accurate state-based specifications from a set of input strings.

3. An extensive evaluation of the proposed technique by applying a tool prototype, called jMiner, to a set of non-trivial open source software packages; the results show the effectiveness of our approach: many meaningful specifications were mined and several bugs were revealed in our experiments.

2. RELATED WORK

Numerous approaches have been proposed to mine specifications from observed execution traces, among which [2] is closest to our approach. The technique in [2] also takes two steps. It first extracts scenarios of *actions* (parametric events in our case) from the observed parametric trace using data-flow dependence information provided by the user. The extracted scenarios are then standardized into strings and fed into the PFSA learner to infer an automaton that describes the usage pattern of the designated APIs. The major difference between [2] and our approach, in addition

[2]Only the first three digits are used in this paper for simplicity; the full hashcode is used in implementation.

to our extension to the PFSA learning algorithm, is the different methods to extract non-parametric trace slices (i.e., strings). In [2], the scenario extraction is based on the user-provided data-flow dependence among events and some *seed events*. In our approach, the parametric trace is sliced according to a user-defined set of parameters, usually a set of interacting classes (Java classes, e.g., in our tool).

Our experiments show that the dependence-based algorithm in [2] often produces incomplete scenarios and results in over-narrowed specifications that may cause false positives in bug detection, which was indicated in the evaluation results in [2]. Our approach, on the contrary, is more effective in identifying meaningful traces and mining accurate specifications. For example, we were able to mine a property about using sockets similar to the one discussed in [2], while the technique in [2] will generate a very narrow specification for the above Collection-Iterator example, as discussed below. Let us suppose that the following trace is observed (the format of the trace follows [2] to facilitate discussion; note that Collection=c1 is equivalent to Collection:c1 in Figure 2):

```
1. update(Collection = c1)
2. createIter(Collection = c1, Iterator = i1)
3. hasNext(Iterator = i1)
4. createIter(Collection = c1, Iterator = i2)
5. hasNext(Iterator = i2)
6. update(Collection = c1)
7. createIter(Collection = c1, Iterator = i3)
```

Also suppose that the data-flow information is: update defines Collection, createIter defines Iterator and uses Collection, and hasNext uses Iterator. Therefore, for example, event 2 depends on event 1 because event 1 defines c1 and event 2 uses c1. We also need to choose a seed event as the starting point for the scenario extraction. In this case, createIter is the only choice since it contains both parameters. Consequently, three scenarios will be extracted: 1 2 3, 1 4 5, and 6 7. *None of them is complete* with regards to the interaction between Collection and Iterator: the first two do not include event 6 and the last one does not include event 1. In fact, it is easy to see that any scenario extracted using the technique in [2] contains only one update event at the beginning. Therefore, the PFSA learner will eventually infer the automaton in Figure 4 from such scenarios (assuming that enough scenarios are collected). This specification is over-restricted comparing with the one in Figure 1, since it allows only one update before createIter and none after createIter.

Moreover, the technique in [2] requires more effort from the user. One needs not only to provide the data-flow dependence and seed events, which require certain knowledge about the target APIs, but also to tune the provided information to achieve an ideal result. In the above example, one may try to overcome the incomplete scenarios by stating that update defines Collection and also uses Collection. Only one scenario will be extracted using this new dependence information: 1 2 3 4 5 6 7, which includes operations on different iterators and will produce confusing results. In our approach, one only needs to provide event definitions that are already needed for logging the execution.

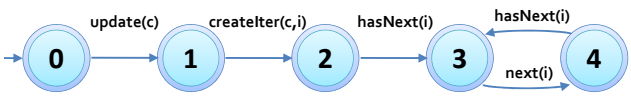

Figure 4: Restricted Collection-Iterator **pattern**

Pradel et al. [22] also presents a technique that infers finite state machines (FSM) from program executions. It first collects a list of relevant receiver-method pairs up to a user-specified level of nested method calls, called *collaboration*. Then, it merges multiple collaborations into a *collaboration pattern* and derives a FSM from it. Since [22] does not have a step similar to our trace slicing, it may infer inaccurate specifications. Let us suppose that the following trace is observed within a method:

```
1. c1.createIter() (returns i1)
2. i1.hasNext()
3. c1.createIter() (returns i2)
4. i2.hasNext()
5. i2.next()
6. i1.next()
```

From this trace, [22] infers a specification that allows two consecutive next, because the interaction of i1 and that of i2 are not separately treated. In our approach, the trace slicer treats these interactions separately and generates a separated trace slice for each interaction. As each interaction is clearly separated after slicing, any learner that infers a property from a set of strings can be employed in our approach. Moreover, [22] cannot infer a specification that spans multiple threads because a separate trace is created for each thread. However, our approach allows events from multiple threads to be combined into one trace, and thus such specifications can be mined.

In addition, FSM-deriving algorithm is different. Although the method used in [22] can generally derive compact FSMs, they may be too permissive. For example, a sequence of method calls a, b, a, c, a will result in a FSM equivalent to $(a(ba|ca)*)$, although the desired specification could be the sequence itself.

Among other mining approaches, [17] also makes use of the PFSA algorithm to infer state-based specifications from observed executions, but it focuses on typestate specifications that involve *only one object* and is essentially non-parametric. [28] and [13] present techniques to mine fixed types of patterns from execution traces: the alternating pattern $((ab)^*)$ or the resource usage pattern $((ab^+c)^*)$. Comparing with our approach, they are not able to mine complex patterns but they do not require any pre-defined symbols to use in the specification. Also, [13] does not consider parameters, and thus it may infer incidental properties from sequences of irrelevant events that happen to match the fixed patterns. [12] extends [13] to allow complex patterns to be inferred, but it still does not utilize parameters. [1] proposes an approach to efficiently infer state-based specifications using frequent closed partial orders (FCPO), but it cannot handle more complicated patterns, like in Figure 1. [21] presents an advanced algorithm to mine extended finite state machines (EFSM), finite state machines extended with state constraints. Conceptually, EFSM subsumes the parametric specification discussed in this paper since a parameter binding can be regarded as a state constraint. However, such generalization of EFSM requires having different sets of states for different parameter bindings in the generated automaton. As we show in the experiments, the number of parameter bindings can be large in practice and may result in a very complicated automaton using EFSM. But it can be beneficial to combine EFSM with our parametric trace slicing framework to handling both parameters and state constraints.

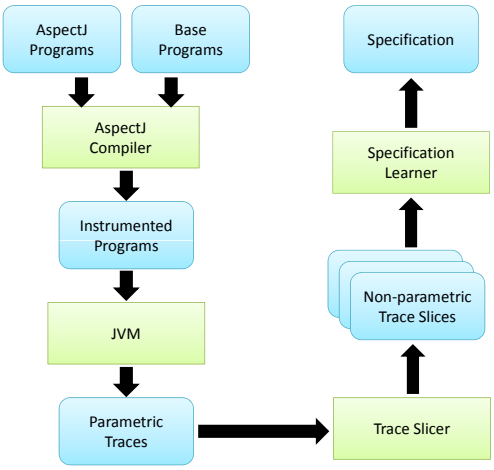

Figure 5: Parametric Specification Mining

Some static analysis-based approaches have also been proposed to infer automata-based specifications, including [24], [25] and [27]. These approaches can avoid the inherently limited coverage of dynamic mining, but they are usually less scalable and create more redundancy in the results. Also, there are many dynamic analysis approaches to infer specifications other than finite state machines. For example, Daikon [10] and [14] infer state predicates, and [6], [19] and [20] mine sequence diagram-based specifications. Comparing with our approach, [20] is not aimed at inferring correct behaviors of interacting objects and detecting bugs. It focuses on capturing how distinct components cooperate in a system. Thus, it neglects concrete objects in each event; it only considers their package names or class names. Due to this difference, it is not suitable for mining specifications from interleaving traces. Moreover, it does not consider repeated events, which makes it unsuitable for mining specifications that include loops such as Figure 1.

3. APPROACH OVERVIEW

Our solution to mine parametric specifications has three stages, as depicted in Figure 5. The first stage is to collect execution traces from the base programs that exercise the uses of the components under mining. Here we assume that the base programs are given, but in practice one may use existing test cases, benchmarks (for example, we used the DaCapo benchmark [5] in our experiments) or some automated tools, such as the model-checker-based approach proposed in [1], as the base programs. To make base programs emit execution traces, we instrument them using AspectJ [18]. AspectJ provides an expressive, pattern-based language to describe *join points* in a program, i.e., points where one may insert actions, giving an elegant way to generate execution traces. For example, Figure 6 shows the AspectJ code used to log method invocations of Collection and Iterator objects. Three *advice* are defined to log three different kinds of methods: the methods related to Collection only, the method involving both Collection and Iterator, and the methods of Iterator. thisJoinPoint.getSignature().getName() returns the name of the invoked method. Such AspectJ programs are automatically generated once a set of target classes is provided by the user. It may be convenient to record all method invocations and to refine the execution traces later using exclusion lists and grouping information (see Discussion below). The AspectJ compiler then weaves the instrumentation into the base programs. The instrumented programs will log a projection of their execution at the specified join points, resulting in parametric traces similar to those in Figure 2.

After the execution trace is collected, it will be sent to the trace slicer and sliced into a set of non-parametric trace slices according to a user-specified set of parameters. One can choose all classes used in the AspectJ program as the set of parameters, e.g., {Collection, Iterator} for Figure 6. One may also choose {Collection} or {Iterator} as the set of parameters if one wants to mine specifications of only one class. In general, it is good practice to log the execution of the base program only once for many components, and then use the resulting trace to mine specifications over different sets of components without running the program again.

The generated non-parametric trace slices are sent to the specification learner, which generates non-parametric finite-state specifications together with equivalent regular expressions (for convenience). The learned specification is then associated with parameter information to output a parametric specification. Thus, the learner *need not be aware* of parameters; this allows us to reuse existing learners and/or ease the task of developing new ones. Although we currently use a refined version of the PFSA-based algorithm (Section 5), the divide-and-conquer solution advocated here is general and other techniques, such as PCFO mining [1], may also be used.

Discussion and Limitations of our Technique. After extensive experimentation with our mining technique as implemented in JMiner (see Section 6), we found that it has two major limitations: 1) it requires the user to provide event definitions, and 2) the learning process is limited to the observed program behaviors. However, we believe that these are inherent limitations of all dynamic mining approaches that aim at the same degree of generality as we do, so, instead of trying to eliminate these limitations in a fully automatic manner, we prefer to provide guidelines and tool support to reduce their negative impact on the user. We next discuss these.

The user-provided event definitions are necessary to instrument the program to emit a parametric trace at runtime and then to slice the generated trace (a careful analysis of other related mining techniques, e.g., [2] and [1], shows that those also rely on events, some even take their availabil-

```
import java.util.*;
public aspect col_iter_Logger {
  TraceWriter w = new TraceWriter("col-iter.log");
  after(Collection c) :
   (call(* Collection+.add*(..)) || call(* Collection+.clear())
   || call(* Collection+.remove*(..))) && target(c) {
     w.log(getThisMethodName(), new Object[]{c});
  }
  after(Collection c) returning(Iterator i) :
   (call(* Collection+.iterator()) && target(c)) {
     w.log("createIter", new Object[] {c, i});
  }
  after(Iterator i) :
   (call(* Iterator+.*()) && target(i)) {
     w.log(getThisMethodName(), new Object[] {i});
  }
  String getThisMethodName() {
     return thisJoinPoint.getSignature().getName();
  }
}
```

Figure 6: AspectJ code for logging uses of Collection and Iterator

ity for granted). The user has flexibility in how to define the events. One approach is to focus on one property at a time, and instrument the program specifically for that property; the drawback of this approach is that one needs to run the program again for each property, which can be quite expensive. Another approach is to run the program once, but instrumented to emit generic events, say one event per method call, and then refine the resulting comprehensive trace for each property of interest. jMiner allows the user to provide *exclusion* and *grouping* information: the former consists of a list of events to be ignored and the latter of a set of events to be grouped into one new event. For example, the grouping input "`update: add remove clear`" groups three operations of Collection into one update event; we used it to transform the original trace logged by the code in Figure 6 into the trace in Figure 2, which resulted in the compact specification in Figure 1. We may further group next and hasNext into a useIter event, which will collapse states 2 and 3 in Figure 1 into one state with a self-cycle.

Although providing the exclusion and grouping information allows the user to achieve the best traces for mining, we want to emphasize that the user interested in spending little or no effort on event definitions can still use our technique: e.g., the Java debugger can be easily configured to output comprehensive traces; then one can run jMiner on those traces. This approach can be particularly useful when one wants to use mining in combination with testing, as we do in Section 6: first mine library properties using programs believed to be correct, then monitor those properties against programs which may possibly misuse the libraries. This process can be automated and give one an automatic way to find bugs.

jMiner also provides a means to detect possible grouping definitions to reduce the user's effort. First, grouping methods using common prefixes or suffixes, such as get and set, are effective in many cases. Sometimes, methods that start with get and is can also be grouped into a use event since they usually are pure methods. The user of jMiner can choose to automatically apply them during the mining process. After an automaton is inferred, events that cause the same transitions are grouped using disjunctions. One can choose to group such events to achieve a more compact specification. For example, the original pattern mined for the Collection-Iterator example in Figure 1 was (add || remove || clear)* (createIter(hasNext next)* hasNext? (add || remove || clear)*)?. jMiner thus reported (add || remove || clear) as a grouping candidate.

As mentioned above, a second limitation of our mining approach is that the quality of the inferred property strictly depends on the observed program execution. For example, the mined specification in Figure 1 may wrongly suggest that two consecutive hasNext events are disallowed; since multiple consecutive hasNext events have not been observed in any of our test programs, the miner had no chance to learn the complete specification. This limitation is shared by all dynamic analysis techniques and we do not address it in this paper. Test case or program generation techniques may ameliorate this limitation, but we did not attempt to use them.

4. SLICING TRACES FOR MINING

Slicing a parametric trace with a set of parameters consists of dispatching events to trace slices corresponding to different bindings of the parameters. Although the intuition is clear, developing efficient and correct slicing algorithms is hard. First, traversing the trace more than once is undesirable due to efficiency concerns. Second, an event may contain an incomplete binding of the given set of parameters and/or irrelevant parameter instances. For example, for the trace in Figure 2, if we choose {Collection} as the set of parameters, a createIter event contains, in addition to a desired Collection object, an irrelevant Iterator object. If we choose {Collection, Iterator} as the set of parameters, an update event contains only a Collection object, leaving the Iterator parameter unbound. Sometimes, no event provides a complete binding for the given set of parameters, meaning that we need to combine parameter information from multiple events to achieve a complete binding; Figure 13 shows such as example. Third, not all parameter instances achieved by combining different events' parameter bindings are meaningful, as will be discussed in Section 4.2. A parametric trace slicing algorithm was proposed in [9] in the context of monitoring, but it turns out that it has several limitations when used for mining.

In this section we introduce a novel slicing algorithm that overcomes the limitations of the one in [9] in the context of mining.

4.1 Parametric Trace Slicing

Our notation and terminology in this section is borrowed from [9].

Definition 1. Let $\mathcal{E}$ be a set of non-parametric events, called **base events** or simply **events**. An **$\mathcal{E}$-trace**, or simply a **trace**, is any finite sequence of events in $\mathcal{E}$, that is, an element in $\mathcal{E}^*$. If event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$ then we write $e \in w$.

Let $[A \to B]$ and $[A \rightharpoonup B]$ denote the sets of total and respectively partial functions from A to B. What follows extends the definition above to parametric events and traces.

Definition 2. **(Parametric events and traces)** Let X be a set of **parameters** and let V_X be a set of corresponding **parameter values**. If $\mathcal{E}$ is a set of base events as defined in Definition 1, then let $\mathcal{E}\langle X\rangle$ denote the set of corresponding **parametric events** $e\langle\theta\rangle$, where e is a base event in $\mathcal{E}$ and θ is a partial function in $[X \rightharpoonup V_X]$. θ is called a **parameter instance** or **parameter binding**. A **parametric trace** is a trace with events in $\mathcal{E}\langle X\rangle$, that is, a word in $\mathcal{E}\langle X\rangle^*$.

For the trace in Figure 2, we have $\mathcal{E}$ = {update, createIter, next, hasNext}, X = {Collection, Iterator} and V_X = {158, 119, ...}. update(Collection:158) and next(Iterator:119) are simplified from update $\langle\theta$(Collection)=158$\rangle$ and next$\langle\theta$(Iterator)=119 $\rangle$, respectively.

Definition 3. Two parameter instances θ and θ' are **compatible** iff for any $x \in \mathsf{Dom}(\theta) \cap \mathsf{Dom}(\theta')$, $\theta(x) = \theta'(x)$. We can **combine** compatible instances θ and θ', written $\theta \sqcup \theta'$, as follows:

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \text{when } \theta(x) \text{ is defined} \\ \theta'(x) & \text{when } \theta'(x) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

θ' is **less informative** than θ, written $\theta' \sqsubseteq \theta$, iff for any $x \in X$, if $\theta'(x)$ is defined then $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$. Let $\theta \sqcap \theta'$ be the most informative θ'' s.t. $\theta'' \sqsubseteq \theta$ and $\theta'' \sqsubseteq \theta'$.

For example, (Collection:158) compatible with (Collection:158, Iterator:119), (Collection:158)$\sqcup$(Collection:158, Iterator:119) is (Collection:158,Iterator:119), (Collection:158)$\sqsubseteq$(Collection:158,Iterator:119), and (Collection:158)$\sqcap$(Collection:158, Iterator:119) is (Collection:158).

Definition 4. **(Trace slicing)** Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ and partial function θ in X, let the **θ-trace slice** $\tau{\restriction_\theta} \in \mathcal{E}^*$ be the non-parametric trace in $\mathcal{E}^*$ defined as follows:

- $\epsilon{\restriction_\theta} = \epsilon$, where ϵ is the empty trace/word, and
- $(\tau\, e\langle\theta'\rangle){\restriction_\theta} = \begin{cases} (\tau{\restriction_\theta})\, e & \text{when } \theta' \sqsubseteq \theta \\ \tau{\restriction_\theta} & \text{when } \theta' \not\sqsubseteq \theta \end{cases}$

Intuitively, a slice $\tau{\restriction_\theta}$ first filters out all the parametric events that are not relevant to the instance θ, and then, for the events relevant to θ, it forgets the parameters in order to build a non-parametric trace. For parameter instance (Collection:263, Iterator:131) in Figure 2, the trace slice is: createIter hasNext. Other events are incompatible with (Collection:263, Iterator:131), e.g., hasNext(Iterator:119) binds Iterator with a different value.

It is easy to implement a simplistic slicing algorithm: first traverse τ to construct all possible parameter instances θ with $\mathsf{Dom}(\theta) = X$, and then traverse τ again to distribute every event to $\tau{\restriction_\theta}$ for different θ according to the condition in Definition 4. However, as mentioned, this is inefficient because τ is scanned twice, causing unnecessary overhead especially when τ is very long.

A better algorithm is given in [9], called $\mathbb{C}\langle X\rangle$, which calculates both all the instances θ above and their corresponding slices in *one online traversal* of τ. Table 1 illustrates a run of algorithm $\mathbb{C}\langle X\rangle$ over the first five events in Figure 2, where $X = \{\mathsf{Collection}, \mathsf{Iterator}\}$. When the first event update(Collection:158) is received, a new trace slice is created for the parameter instance (Collection:158) and the event update is added to that slice. The second event createIter(Collection:158, Iterator:119) also results in a new trace slice which is initialized with the trace slice for (Collection:158) because (Collection:158) is less informative than (Collection:158, Iterator:119). Then the event createIter is appended to the new slice. A trace slice for (Iterator:119) is created at the third event hasNext(Iterator:119) with the initial value ϵ because no other parameter instance is less informative than it. The event hasNext is then appended to both slices corresponding to (Collection:158, Iterator:119) and (Iterator:119). The fourth event next(Iterator:119) does not lead to the creation of a new trace slice and next is appended to both slices corresponding to (Collection:158, Iterator:119) and (Iterator:119). Two new trace slices are created at the fifth event update(Collection:148), namely, one for (Collection:148) and the other for (Collection:148, Iterator:119). The former is initialized to be ϵ and the latter is initialized using the existing slice for (Iterator:119).

To compute all the slices in one pass, $\mathbb{C}\langle X\rangle$ also generates trace slices for parameter instances θ with $\mathsf{Dom}(\theta) \subset X$; e.g., slices for (Collection:158) and (Iterator:119) in Table 1. These slices are used for intermediate purposes only; e.g., the slice for (Collection:158) is used to initialize the slice for (Collection:158, Iterator:119).

$\mathbb{C}\langle X\rangle$ was proposed in the context of runtime monitoring [9], which holds different assumptions from specification mining.

4.2 Restricting Slices to Connected Parameter Instances

Algorithm $\mathbb{C}\langle X\rangle$ [9] may generate slices that correspond to meaningless parameter instances. For example, in Table 1, the parameter instance (Collection:148, Iterator:119) created at the fifth event is meaningless: Iterator:119 is an iterator over Collection:158, totally unrelated to Collection:148. In the context of monitoring, such spurious parameter instances can be omitted by the underlying monitor using knowledge extracted from the specification. In the context of mining, however, the trace slices generated for the spurious parameter instances become "noise" and thus may reduce the accuracy of the inferred specification. For example, when the sixth event hasNext(Iterator:119) in Figure 2 is received, the trace slice for (Collection:148, Iterator:119) becomes hasNext next update hasNext. If this trace slice is included in the input to the specification learner, the resulting specification would allow the collection to be updated while using the iterator, which would be incorrect. To address this problem, we introduce the concept of *connected parameter instances*, which effectively remove meaningless trace slices.

Definition 5. Given $\tau \in \mathcal{E}\langle X\rangle^*$, we define **$\tau$-connectedness** of parameter instances θ as follows: 1) if $e\langle\theta\rangle \in \tau$ then θ is τ-connected; and 2) if θ_1, θ_2 are τ-connected, compatible, and $\theta_1 \sqcap \theta_2 \neq \bot$, then $\theta_1 \sqcup \theta_2$ is also τ-connected.

Hence, a parameter instance is τ-connected iff it is formed by combining parameter instances of events in τ that share parameter bindings. For example, in Table 1, (Collection:158, Iterator:119) is connected because of the first createIter event; (Collection:148, Iterator:119) is not connected because Collection:148 and Iterator:119 do not occur in the same event. This concept is motivated by the following observation: in most cases we are interested in mining specifications for a set of *interacting objects*; if two objects appear in the same event (as concrete values bound to the event's parameters) then they interact with each other. Therefore, all the objects contained in a connected parameter instance directly or indirectly interact with one another. Experiments using our mining technique, partly discussed in Section 6, show that passing only the trace slices corresponding to connected parameter instances to the specification learner (and discarding the other trace slices) effectively removes noise in the mining process, resulting in accurate specifications.

Input: X, $\tau = e_1\langle\theta_1\rangle, e_2\langle\theta_2\rangle, \ldots, e_n\langle\theta_n\rangle$
Map: $\Delta \in [[X \rightharpoonup V_X] \rightharpoonup \mathcal{E}^*]$, $\mathcal{U} \in [[X \rightharpoonup V_X] \rightarrow \mathcal{P}_f([X \rightharpoonup V_X])]$
Set: $\Theta \subseteq [X \rightharpoonup V_X]$, $\Psi \subseteq [X \rightharpoonup V_X]$
Initial: $\mathcal{U}(\theta) \leftarrow \emptyset$ for any $\theta \in [X \rightharpoonup V_X]$, $\Delta(\bot) \leftarrow \epsilon$, $\Theta \leftarrow \Psi \leftarrow \emptyset$

Function MAIN()

1 for $i \leftarrow 1$ to n do HANDLEEVENT($e_i\langle\theta_i\rangle$)
2 CONSTRUCTCONNECTED()

Function HANDLEEVENT($e\langle\theta\rangle$)
Function DEFINENEW(θ)
Function CONSTRUCTCONNECTED()

Figure 7: $\mathbb{M}\langle X\rangle$: Slicing with connected parameter instances

It is easy to implement a slicer as above on top of $\mathbb{C}\langle X\rangle$: all one needs to do is to include an additional check for connectedness of the corresponding parameter instance before outputting each slice. We have implemented such a slicer but, unfortunately, it turned out to be rather slow, sometimes unusable, in experiments (see Table 2). The reason

update(Collection:158)	createIter(Collection:158, Iterator:119)	hasNext(Iterator:119)
(): ϵ (Collection:158): update	(): ϵ (Collection:158): update (Collection:158, Iterator:119): update createIter	(): ϵ (Collection:158): update (Collection:158, Iterator:119): update createIter hasNext (Iterator:119): hasNext

next(Iterator:119)	update(Collection:148)	...
(): ϵ (Collection:158): update (Collection:158, Iterator:119): update createIter hasNext next (Iterator:119): hasNext next	(): ϵ (Collection:158): update (Collection:158, Iterator:119): update createIter hasNext next (Iterator:119): hasNext next (Collection:148): update (Collection:148, Iterator:119): hasNext next update	...

Table 1: A run of the trace slicing algorithm $\mathbb{C}\langle X\rangle$ (top table first, followed by the bottom table).

```
1  if Δ(θ) undefined then
2      if θ ∉ Θ then DefineNew(θ)
3      if θ ∉ Ψ then Ψ ← Ψ ∪ {θ}
4      Δ(θ) ← ϵ
5      foreach θ_max ⊏ θ in reversed topological order do
6          foreach θ_comp ∈ U(θ_max) compatible with θ do
7              if θ_comp ⊔ θ ∉ Θ then DefineNew(θ_comp ⊔ θ)
8              if θ_comp = ⊔_{i=1}^{k} θ_i ∧ ∀i ∈ 1,k.(θ_i ∈ Ψ ∧ θ_i ⊓ θ ≠ ⊥)
               then
9                  Ψ ← Ψ ∪ {θ_comp ⊔ θ}
10 Δ(θ) ← Δ(θ) e
```

```
1  Θ ← Θ ∪ {θ}
2  foreach θ' ⊏ θ do U(θ') ← U(θ') ∪ {θ}
```

for the lack of performance of this simple slicer is that, even though it only reports the slices corresponding to connected parameter instances, internally it still generates the trace slices for all possible parameter instances. Maintaining all trace slices can significantly increase the space requirements. In many of our experiments there were only about 5-10% of the parameter instances which were connected (see Table 2). Note that the space overhead grows faster than the number of parameter instances, because an event can belong to multiple trace slices. Additionally, maintaining all trace slices increases the time overhead because each event should be dispatched to more trace slices.

For better performance, we developed a new trace slicer, named $\mathbb{M}\langle X\rangle$ (from jMiner), which avoids the construction of unnecessary trace slices. Figure 7 shows our slicer. $\mathbb{M}\langle X\rangle$ has two stages: (1) it first processes the entire parametric trace, event by event, and while doing so it incrementally updates two internal sets of parameter bindings, Θ and Ψ; (2) then it constructs all the trace slices corresponding to connected parameter instances. During the first stage, $\mathbb{M}\langle X\rangle$ constructs (and saves in Δ) trace slices only for parameter instances that actually occur in observed events (i.e., θ_1, θ_2, ..., θ_n). Θ holds all the combinations of compatible parameter instances, but trace slices for them are *not* constructed. For example, unlike in $\mathbb{C}\langle X\rangle$ (see Table 1), the fifth event update(Collection:148) in Figure 2 does not introduce two new trace slices. At any given moment, $\mathcal{U}$ (from "up") holds for each instance θ all its more informative existing compatible instances: $\mathcal{U}(\theta) = \{\theta' \in \Theta \mid \theta \sqsubset \theta'\}$. After the loop in Main terminates, Θ contains all possible compatible parameter instances, Ψ contains all connected parameter instances, and Δ contains trace slices for parameter instances occurring in events. The function ConstructConnected goes through every connected parameter instance θ in Ψ and constructs its trace slice by merging existing slices for θ's sub-instances. We leave out the code for the function MergeTraces because it is trivial.

```
Set: μ ⊆ E*
1  foreach θ ∈ Ψ ∧ Dom(θ) = X do
2      μ ← ∅
3      foreach θ' ⊑ θ do if Δ(θ') defined then
       μ ← μ ∪ {Δ(θ')}
4      Δ(θ) ← MergeTraces(μ)
```

Consider the trace "$e_1(b_1, c_1)\ e_2(a_2, d_2)\ e_3(a_2, b_1)$"; for simplicity, we only write the parameter values, i.e., $e_1(b_1, c_1)$ means e_1 is parametric in b and c, which are instantiated to b_1 and c_1, resp. After e_1 is processed, $\Theta = \Psi = \{\langle b_1,c_1\rangle\}$. After e_2 is processed, $\Theta = \{\langle b_1,c_1\rangle, \langle a_2,d_2\rangle, \langle a_2,b_1,c_1,d_2\rangle\}$ and $\Psi = \{\langle b_1,c_1\rangle, \langle a_2,d_2\rangle\}$. Finally, after e_3 is processed, $\Theta = \Psi = \{\langle b_1,c_1\rangle, \langle a_2,d_2\rangle, \langle a_2,b_1\rangle, \langle a_2,b_1,c_1\rangle, \langle a_2,b_1,d_2\rangle, \langle a_2,b_1,c_1,d_2\rangle\}$. There is only one instance in Ψ binding all parameters, with slice $e_1e_2e_3$.

Theorem 1. *After running $\mathbb{M}\langle X\rangle$ on $\tau \in \mathcal{E}\langle X\rangle^*$, the next hold:*

1. *Parameter instance $\theta \in \Theta$ is connected iff $\theta \in \Psi$; and*
2. *If $\theta \in \Psi$ and $Dom(\theta) = X$, then $\Delta(\theta) = \tau{\restriction_\theta}$.*

This theorem also tells how retrieve all the trace slices corresponding to connected parameter instances: $\{\Delta(\theta) \mid \theta \in \Psi\}$.

5. LEARNING SPECIFICATIONS FROM TRACES

We next discuss our specification learning algorithm that infers a deterministic finite automaton (DFA) together with an equivalent regular pattern from non-parametric trace slices. The proposed learning process is illustrated in Figure 8. It consists of three components: a PFSA learner, an automaton refiner, and a regular pattern generator. The regular pattern generator uses the Brzozowski method [7] to generate equivalent regular patterns from the inferred DFA. The generated regular pattern is then simplified using a set of rules (not discussed here). The algorithm produces a reasonably compact (but not necessarily minimal) pattern that can be easily understood, facilitating documentation and program understanding.

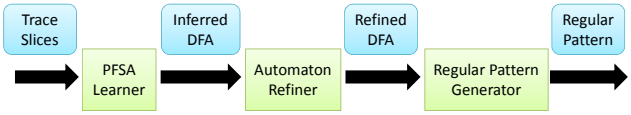

Figure 8: Learning Process

5.1 PFSA Learner

A PFSA learner [11] takes a set of strings (here non-parametric trace slices) as input and infers a finite state automaton, where each node represents the state of the involved components and each edge represents an event. The inferred automaton typically accepts more than the given set of strings, since the PFSA learner usually generalizes during the learning process. Several PFSA learning approaches have been proposed [11]; we here adopt the sk-strings algorithm [23], since it performs well at inferring small automata.

The sk-strings PFSA learner first constructs an automaton that precisely accepts the input set of strings. Each transition is then annotated with a frequency, saying how many times that transition was observed. The sk-strings algorithm then generalizes by merging states which are *sk-equivalent*: two states as sk-equivalent iff corresponding sets of bounded strings (ones that are frequently generated from each of the two states) are matched. As a result of this approximation, two states can be merged even when they are not equivalent. This makes it possible for the inferred automaton to accept not only the input strings but also other "similar" strings. The reader is referred to [23] for more details. Figure 9 shows the automaton inferred by the sk-strings PFSA learner for our Collection-Iterator example (Figure 1 shows the actual desired specification).

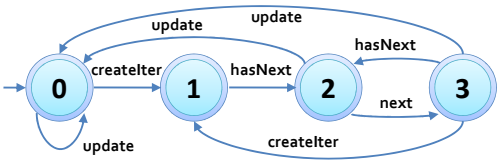

Figure 9: DFA inferred by the PFSA learner

5.2 Automaton Refiner

Function MAIN()

Input : automaton $A = (S, \mathcal{E}, i, \delta : [S \times \mathcal{E} \rightharpoonup S], F)$, traces $T \subseteq \mathcal{E}^*$
Output: automaton A_r
Locals : automaton $A' = (S', \mathcal{E}, i', \delta', F')$, state s, s', transition function δ_r

1 $A' \leftarrow$ EXPAND(A)
2 $\delta_r \leftarrow \perp$
3 foreach $\tau \in T$ do
4 $s \leftarrow i'$
5 foreach $e \in \tau$ do
6 $s' \leftarrow s$; $s \leftarrow \delta'(s, e)$; $\delta_r(s', e) \leftarrow s$
7 if $\delta_r = \delta'$ then goto 8
8 $A' \leftarrow (S', \mathcal{E}, i', \delta_r, F')$
9 $A_r \leftarrow$ MERGEIDENTICALSTATES(A')

Function EXPAND(A)

Although the PFSA learner approximations are typically desirable in many application domains, the resulting automata turned out to often be overly-general in our domain of mining specifications from execution traces, in that the mined automata accept a large number of undesirable traces. For example, the trace createIter hasNext next createIter is accepted by the automaton in Figure 9, but it is impossible to occur in any program execution (only one createIter event can be observed in any interaction between a collection and an iterator). An over-generalized specification undermines the effectiveness of its applications. For example, it may cause misunderstanding of the system behaviors when used for reverse engineering, or miss potential errors when used for program testing and verification. Also, an over-generalized DFA often produces a more complex regular expression (e.g., the one in Figure 9), increasing the difficulty of human inspection.

We next propose an approach to refine the PFSA-inferred automata. Our DFA-refiner may have a broader range of applications, but we have only experimented with it in the context of specification mining, where it appears to work very well. The overall goal of our refiner is to eliminate transitions caused by over-generalizations while keeping desirably generalized transitions. An obvious step is to remove all the transitions that are never taken by any of the trace slices that were provided as the input of the PFSA learner. For example, the createIter transition from state 3 to state 1 in Figure 9 is never taken (no trace can create the same iterator twice), so it can be safely removed. Unfortunately, this is not enough. Indeed, the remaining DFA in Figure 9 still accepts infeasible traces containing two or more createIter transitions, e.g., "createIter hasNext update createIter ...". This suggests that what is really needed is to "break" the spurious loops, such as the loop containing states 0,1,2 in Figure 9. Indeed, since no trace slice provided as input to the learner is observed to take a given loop in the learned automaton, there is no reason to assume that other trace slices would take it.

Figure 10 shows our algorithm $\mathbb{R}$ (from Refiner). $\mathbb{R}$ first expands the automaton using the Expand function, which clones each state a number of times so that each incoming edge has its own clone. For the PFSA-inferred automaton, let S be its set of states, $\mathcal{E}$ be the set of base events as de-

Input : automaton $A = (S, \mathcal{E}, i, \delta, F)$
Output: automaton $A' = (S', \mathcal{E}, i', \delta', F')$
Locals : integer n, set of states D, map $\gamma : S \rightarrow 2^{S'}$
Initial : $S' \leftarrow \emptyset$, $F' \leftarrow \emptyset$, $\delta' \leftarrow \perp$

1 foreach $s \in S$ do
2 $n \leftarrow$ COUNTINCOMINGEDGES(s, A)
3 if $s = i$ then $n \leftarrow n + 1$
4 $D \leftarrow$ GETFRESHSTATES(n)
5 $S' \leftarrow D \cup S'$
6 $\gamma(s) \leftarrow D$
7 foreach $s \in S$ do
8 foreach $s' \neq s \in S$ *s.t.* $\delta(s', e) = s$ *for some* e do
9 $s'' \leftarrow$ PICKONEWITHNOINCOMINGEDGE($\gamma(s), \delta'$)
10 foreach $s''' \in \gamma(s')$ do $\delta'(s''', e) = s''$
11 if $s \in F$ then $F' \leftarrow F' \cup \gamma(s)$
12 if $s = i$ then
 $i' \leftarrow$ PICKONEWITHNOINCOMINGEDGE($\gamma(s), \delta'$)
13 return A'

Figure 10: Automata refining algorithm $\mathbb{R}$

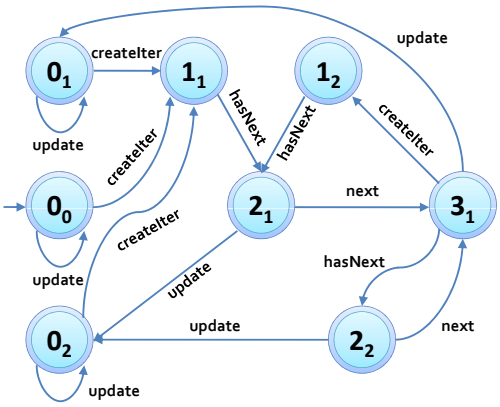

Figure 11: Expanded automaton of Figure 9

fined earlier, and $\delta \in [S \times \mathcal{E} \rightharpoonup S]$ be its transition function. If $\delta(s, e) = s'$ for some $s \neq s'$ then e is *an incoming edge* to s'; by assumption, the initial state has a default incoming edge (line 3 in Expand). Hence, state 0 in Figure 9 has three incoming edges, namely a default one to the initial state, one from state 2 and another from state 3. If state s has n incoming edges then n new states are generated for the new automaton and we keep the mapping from s to the corresponding set of newly created states in γ (lines 4 to 6 in Expand). Function Expand then builds transitions in the new automaton (lines 7 to 12): if $\delta(s', e) = s$ is a transition in the PFSA-inferred automaton and $s \neq s'$ then we choose a state s'' from $\gamma(s)$ with no incoming edges at this point and add transitions from every state in $\gamma(s')$ to s''. If s is a final state then all states in $\gamma(s)$ are also final; if s is the initial state then we choose a state from $\gamma(s)$ with no incoming edges as the new initial state. This way, the PFSA-inferred automaton is expanded to an *equivalent automaton* in which every state has a set of incoming edges corresponding to one incoming edge in the original automaton. For example, Figure 11 shows the expanded automaton of the one in Figure 9. The expanded automaton has the benefit that it separates out the various loops in the original automaton, so that it makes easier to see which ones are taken or not by the trace slices provided as input to the learner.

$\mathbb{R}$ then traverses the expanded automaton using the trace slices and marks the transitions used in the traversal (lines 3 to 9). After all the trace slices are applied, $\mathbb{R}$ removes the unmarked transitions from the expanded automaton. For example, Figure 12 shows the automaton after $\mathbb{R}$ traverses the expanded automaton in Figure 11. Three edges are removed in Figure 12, namely, from 0_1 to 1_1, from 0_2 to 1_1, and from 3 to 1_2 because they are not traversed in any trace slice. The reduced automaton is then compressed by merging states that have the same outgoing transitions and by removing those that have no incoming transitions. For example, state 1_2 in Figure 12 is removed, states 0_1 and 0_2 are merged, and states 2_1 and 2_2 are also merged. Finally, we add back the parameter information removed during the trace slicing process, thus obtaining our mined parametric specification, e.g., the one in Figure 1.

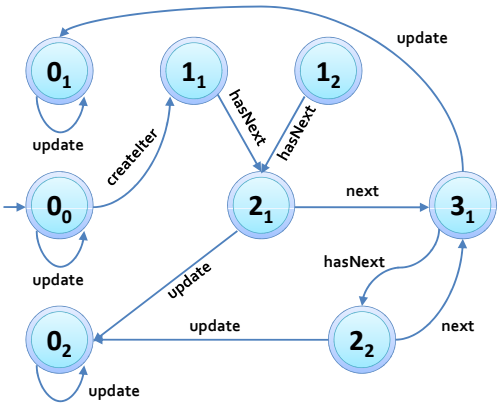

Figure 12: Reduced automaton of Figure 11

THEOREM 2. *With the notation in Figure 10, if $T \subseteq \mathcal{L}(A)$ and A' is the automaton after running $\mathbb{R}$, then $T \subseteq \mathcal{L}(A') \subseteq \mathcal{L}(A)$.*

6. EVALUATION

We have implemented our mining approach in a tool for Java, named JMINER, and evaluated it extensively. The evaluation results are encouraging: JMINER was able to infer many meaningful specifications for various popular programs; we also found several tricky bugs in some well-developed open-source programs by monitoring their executions against the mined specifications using the runtime monitoring tool JavaMOP [8]. We selected for discussion some results from our mining experiments, summarized in Table 2. The first column gives the target classes for which we inferred specifications, together with the software to which they belong. These packages were chosen because of their popularity and diversity. Our experiments cover Java IO library, Java Util library for basic data-structures, Apache Lucene [4] library for text search, JFreeChart [15] library for displaying charts, and Apache JAMES [3].

As mentioned in Section 2, our approach requires less user input than similar mining approaches. However, one still needs to provide event definitions, either manually or automatically. We believe that providing event definitions is an application-specific task which is orthogonal to our mining technique, so we prefer not to advocate any particular way to produce parametric events; we currently use AspectJ for this purpose, but one can also use other program instrumentation means to achieve the same results. In many cases one would like to automate the extraction of events as much as possible. One can, of course, record all possible events, such as all method calls and all variable updates, but then one still needs to filter out such detailed traces to only learn properties about events of interest. There is probably no general recipe to determine what "events of interest" means. We next explain our particular way to obtain such events, which may also work in other contexts.

We automatically scan *unit test cases*, which are typically small and come with the packages for which we want to mine specifications, and hereby collect interacting classes and methods; then we mine properties about parametric traces with events emitted only by those interacting classes and methods, produced by executing large and supposedly correct programs (the unit tests are typically *not* good candidates for producing traces because many of them are aimed at checking how exceptional cases are handled and thus the generated traces represent wrong behaviors). This simple and automatic instrumentation approach builds upon the following hypothesis: *since unit tests are written to check the behavior of tightly interacting objects, there is a high chance that the methods and classes involved in such inter-*

Target classes (software)	Base programs	Trace lengths	Trace slices		Analysis time (seconds)			Mined spec.
			All	Connected	$\mathbb{C}\langle X\rangle$	$\mathbb{M}\langle X\rangle$	learning	
Reader,Writer (Java IO library)	DaCapo (antlr, bloat, chart, eclipse, fop, hsqldb)	9000	1143∼2285	1143∼2285	4	3	< 1	4
Collection,Map,Iterator (Java Util library)	DaCapo (antlr, bloat, chart, eclipse, fop, hsqldb)	50000	79∼90775	19∼4253	757	10	< 1	4
Document,IndexWriter (Apache Lucene)	DaCapo (luindex, lusearch)	29000	792∼11905	792∼7186	> 10000	2090	< 1	3
Plot,JFreeChart,Listener (JFreeChart)	JFreeChart tests	8900	14∼244	14∼42	1	< 1	< 1	6
CommandHandler,SMTPSession (Apache JAMES)	JAMES tests	300	11∼29	11∼29	< 1	< 1	< 1	2

Table 2: Summary of experiments on jMiner

action obey a specification. For example, the following is a test case coming with the Java Util library:

```
List<Integer> list = new ArrayList<Integer>();
for (int i = 0; i < LENGTH; i++)
  list.add(i);
try {
  for (int i : list)
    if (i == LENGTH - 2)
      list.remove(i);
} catch(ConcurrentModificationException e) {
  return;
}
```

Since the Java compiler translates the loops into iterator, hasNext and next, this unit test says that there is an interaction involving the methods Collection.add, Collection.remove, Collection.iterator, Iterator.hasNext and Iterator.next. The events corresponding to these methods are sufficient to mine the specification in Figure 1.

The second column of Table 2 shows the base programs used to generate the execution traces for mining. They include DaCapo[5], a Java performance benchmark suite, and existing tests for JFreeChart and JAMES. The DaCapo benchmark suite contains 11 programs. Some target classes, e.g., Reader and Writer, are used by several programs in DaCapo, In these cases, we used the first half (in alphabetical order) of the DaCapo programs, whose names are given in parentheses, as the base programs, and then monitored the rest against the mined specifications (see Section 6.2).

The third column of Table 2 gives the average lengths of logged parametric traces. The fourth column shows the number of non-parametric trace slices for all (both connected and unconnected) parameter instances computed from the logged parametric traces, and the fifth column shows the number of connected ones. The numbers are given as ranges because the number of trace slices is different from one set of parameters to another. In several cases, only a small percentage of parameter instances are connected, showing the benefit of $\mathbb{M}\langle X\rangle$ over $\mathbb{C}\langle X\rangle$. The sixth and seventh columns give the average time for slicing.[3] $\mathbb{M}\langle X\rangle$ always outperforms $\mathbb{C}\langle X\rangle$; when the number of parameters is larger than one and the parametric traces are long, $\mathbb{M}\langle X\rangle$ is typically orders of magnitude faster than $\mathbb{C}\langle X\rangle$. The "$> 10000$" means that $\mathbb{C}\langle X\rangle$ did not finish in 3 hours. The eighth column shows the average time for learning: our learning algorithm is efficient and always finished in less than one second. Indeed, most of the time is spent producing the right trace slices, not to learn the property. The last column in Table 2 gives the number of mined properties which have between 2 and 10 states when minimized. Our goal in these experiments was to read and understand the mined specifications, in order to evaluate the effectiveness of jMiner. The properties with many states tend to be specific to particular uses of libraries in the base programs rather than to the libraries themselves, so jMiner currently discards them.

[3]All the experiments were carried out on a Ubuntu Linux 7.10 machine with 1.5GB RAM and a Pentium 4 2.66GHz processor.

6.1 Mined Specifications

Due to space constraints, we only briefly list some interesting cases here. More results can be found on jMiner's webpage [16].

Java Util Library: Collection, Map and Iterator. The following properties were mined (see [16] for formal specifications):

1. When using Iterator, one should start with hasNext and then call next and hasNext alternatively:
   createIter hasNext (next hasNext)*.
2. If an iterator is created from a collection, the collection should not be changed when the iterator is being used (Figure 1).
3. If a collection is created from the set of keys or values in a map, the collection should not be changed:
   update* getSet createIter.
4. If an iterator is created from a collection that represents the set of keys or values in a map, the map should not be changed when the iterator is being used (Figure 13).

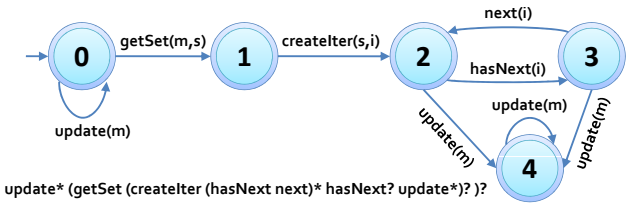

Figure 13: Map-Collection-Iterator **usage pattern**

Property 3 is more restrictive than the library's informal specification: Java allows for removal operations in the collection that represents the key/value set of a map. However, no programs used in our experiments made such operations, resulting in this more restricted specification. Property 4 involves three parameters that are connected indirectly (no events cover all three parameters at the same time), which cannot be mined using other approaches.

JFreeChart: Chart and Plot. JFreeChart is a Java chart library for drawing complex charts. Our evaluation focused on three core classes/interfaces in the JFreeChart package,

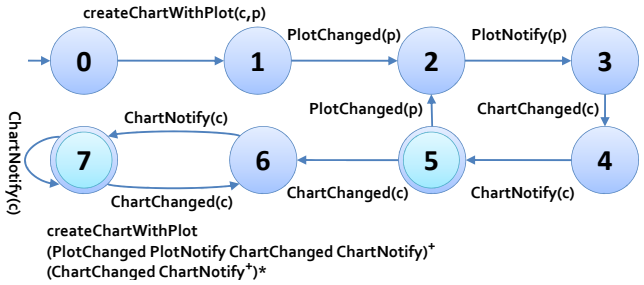

Figure 14: Chart-Plot **pattern in JFreeChart**

namely, JFreeChart, the main entry of the package, Plot, that draws charts, and Listener, that is the interface used for the listener pattern. Both the JFreeChart and Plot classes provide a large number of methods, resulting in complicated specifications. Therefore, we grouped the methods according to their prefixes and/or suffixes to reduce the number of events and simplify the mined specifications.

Figure 14 shows a mined specification involving JFreeChart and Plot. The ChartChanged and PlotChanged methods notify JFreeChart objects and Plot objects, respectively, that some related components have changed. The ChartNotify and PlotNotify methods are used by JFreeChart objects and Plot objects, respectively, to notify other related components. Also, createChartWithPlot is the creation of a JFreeChart object using a Plot object. The desired property is if a chart is created using a plot, then whenever the plot is changed, the chart will be notified and changed accordingly but not the other way around; also, every time a plot/chart is changed, it will send out notifications. Although the mined property implies this pattern, it is more restrictive as it prevents PlotChanged after two consecutive ChartChanged. Such a restriction was learned because it was observed for all test cases we used.

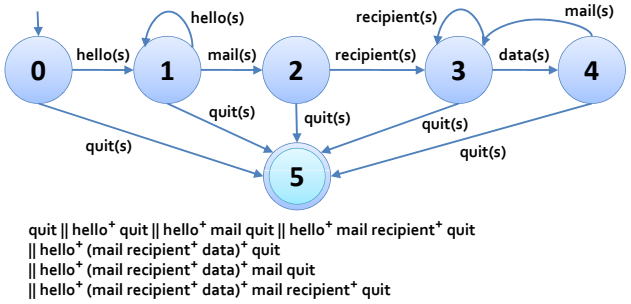

Figure 15: CommandHandler **pattern in Apache JAMES**

Apache JAMES: CommandHandler. We applied our tool to the CommandHandler interface and the SMTPSession class in JAMES' SMTPServer package. A few meaningful specifications were mined, but we only focus on the specification related to CommandHandler here. CommandHandler defines the core interface to handle commands from the mail client. Its mined specification, as showed in Figure 15, precisely describes the usual transaction in SMTP. hello is the command to initiate a mailing session, and quit terminates the session, which can occur at any point. mail starts a mail transaction and sets the sender of an email, recipient adds one or more recipients of the email, and data gives the content of the email and confirms the transaction. After one transaction is done, another transaction can start within a mailing session, which allows the transition from state 4 to state 3.

It is worth noting that, unlike other examples where the method name is used as the event id and the target object and arguments are used as event parameters, we used the class name of the target CommandHandler object, e.g., QuitCmdHandler, as the event id and the argument of the onCommand() method in the CommandHandler interface as the event parameter, which is an SMTPSession object representing a mailing session. It is easy to achieve such changes in our approach not only because AspectJ provides the needed programming capability but also because the parametric slicing algorithm and the learning algorithm in our approach do not depend on any specific meaning of the event and the trace. This way, our approach allows the user to apply domain knowledge in the mining process to achieve better results.

6.2 Problems Revealed Using Mined Specifications

classes	testing programs	detected errors
Reader,Writer	DaCapo (jython, luindex, lusearch, pmd, xalan)	1
Collection,Map, Iterator	DaCapo (jython, luindex, lusearch, pmd, xalan)	1
Document, IndexWriter	SCAN	1
Plot,JFreeChart, Listener	DaCapo (chart)	0

Table 3: Monitoring results of mined specifications.

We used the mined specifications (except for those about Apache JAMES because we did not find suitable testing programs) to monitor another set of programs. The monitoring was done using JavaMOP [8], a runtime monitoring tool for Java. Table 3 shows the results of the monitoring experiments. The first two columns show the classes to monitor and the monitored programs, respectively. In addition to the DaCapo benchmark, we used SCAN [26], a desktop content search engine using Lucene library. The third column gives the number of errors detected. *No false positives* were reported, that is, all the problems reported are actual problems in the programs and none of them were caused by over-narrowed specifications as in [2], showing the effectiveness of our technique.

The first error is caused by the misuse of Iterator in the pmd benchmark in DaCapo. JavaMOP reported a violation of a mined pattern for Iterator, namely, createIter (hasNext next)* hasNext?, in the execution of the benchmark, showing that the program tried to fetch the first element in an iterator without checking hasNext. A further inspection confirmed our observation and showed that this is a potential bug in the program, which has been fixed in a newer version. The second problem, which was mentioned in [8], is related to the uses of Writer in the Xalan benchmark. The execution of the Xalan benchmark tried to use a writer even after it is closed, violating a mined specification of Writer, namely, createWriter write* closeWriter.

The third problem is even more interesting because it revealed *an error in the base program* instead of the tested program. It is related to the uses of the Lucene library for text indexing and search. In our experiments, we chose two main classes, namely, Document and IndexWriter, from Lucene to infer possible specifications. Document is the unit of index-

ing and search. IndexWriter creates and maintains an index. To create indexing information for a file, a Document object is first created for the file, then it is added to an IndexWriter object that writes the indexing information to the index database later. We used the luindex and lusearch benchmarks in DaCapo as the base programs to mine specifications. The mined property is (updateDoc* addDocToWriter)$^+$, in which updateDoc is updating the document and addDocToWriter is adding the document to an index writer. This specification was then monitored against the execution of SCAN and caused a violation. Because we had little knowledge about Lucene, we applied jMiner to SCAN and achieved the specification in Figure 16, in which flushWriter and closeWriter are flushing and closing the index writer, respectively[4]. By comparing these two specifications, it is obvious that the luindex and lusearch benchmarks created and modified indexing information but did not flush them before exit, probably relying on the Java garbage collector to close the writer and write out the modified information. However, according to Lucene's API specification, such behavior could lead to loss of information in the writer, implying a potential bug.

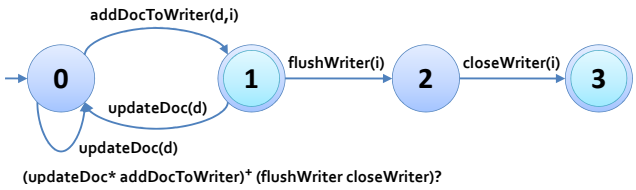

Figure 16: Document-IndexWriter **pattern mined from SCAN**

Also, during the inspection process, we noticed that the transition from state 1 to state 0 in both specifications appears to be redundant, because it simply enforces the Document object to be re-added into the underlying IndexWriter after some changes are made in the document. A further inspection in the Lucene's documentation shows that it is indeed a required pattern for using IndexWriter: the changes on the Document object are *not visible* to the underlying IndexWriter if they are made after the document is added to the IndexWriter. However, this property is not very intuitive and hidden in the informal description of the library, making it easy to be omitted by users who are not familiar with the library. A clear and rigorous specification, like the one in Figure 16, will help anyone to use the library correctly. It can also be used in testing or program verification to detect related subtle errors about using the library.

7. CONCLUSION

This paper presented an approach to mine parametric state-based specifications from observed execution traces. The approach is based on a general framework that generates non-parametric trace slices from parametric traces, thus allowing one to apply mining techniques that do not take parameters into account. A PFSA-based learner to infer compact and precise state-based specifications from non-parametric trace slices was developed within the parametric framework. The approach has been implemented in a tool for Java. Our experiments show that our technique is effective in mining specifications that involve one or more parameters for different types of programs. The results are usually compact, precise and easy to understand, facilitating human inspection and program analysis. Monitoring of the mined specifications revealed problematic behaviors in several open source programs under testing, showing the usefulness of our approach.

[4]We used the same events in mining and monitoring of both programs; therefore, a flushWriter event caused a violation of the property (updateDoc* addDocToWriter)$^+$ since the DFA does not accept the event.

8. REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *FSE'07*, 2007.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specification. In *PLDI'02*, 2002.

[3] http://james.apache.org/.

[4] http://lucene.apache.org/.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, McKinley, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, 2006.

[6] L. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Trans. Software Engineering*, 32(9):642–663, 2006.

[7] J. A. Brzozowski. Derivatives of regular expressions. *Journal of ACM*, 11(4):481–494, 1964.

[8] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *OOPSLA'07*, 2007.

[9] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *TACAS'09*, 2009.

[10] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE'00*, 2000.

[11] J. Feldman and A. Biermann. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–596, 1972.

[12] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *FSE'08*, 2008.

[13] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE'08*, 2008.

[14] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02*, 2002.

[15] http://www.jfree.org/jfreechart.

[16] http://fsl.cs.uiuc.edu/jMiner.

[17] P. Joshi and K. Sen. Predictive typestate checking of multithreaded Java programs. In *ASE'08*, 2008.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP'01*, 2001.

[19] D. Lo and S. Maoz. Mining scenario-based triggers and effects. In *ASE'08*, 2008.

[20] D. Lo and S. Maoz. Mining hierarchical scenario-based specifications. In *ASE'09*, 2009.

[21] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE'08*,

2008.

[22] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE'09*, 2009.

[23] A. V. Raman and J. D. Patrick. The sk-strings method for inferring PFSA. In *ICML'97*, 1997.

[24] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE'07*, 2007.

[25] M. K. Ramanathan, A. Grama, and S. Jagannathan. Specification inference using predicate mining. In *PLDI'07*, 2007.

[26] http://scan.sourceforge.net/.

[27] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA'07*, 2007.

[28] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE'06*, 2006.

APPENDIX

Proof of Theorem 1

In cases where there is no ambiguity, τ-*connected* will be referred to with term *connected* in this section.

LEMMA 1. $\mathcal{U}(\theta) = \{\theta' \mid \theta \sqsubset \theta' \text{and } \theta' \in \Theta\}$ *for all* $\theta \in [X \rightharpoonup V_X]$ *before and after each execution of* DEFINENEW. *Informally,* $\mathcal{U}(\theta)$ *contains all parameter instances that are strictly more informative than* θ.

PROOF. Based on the way that $\mathbb{M}\langle X\rangle$ initializes, for any $\theta \in [X \rightharpoonup V_X]$ we have $\mathcal{U}(\theta) = \emptyset$. Since we also have $\Theta = \emptyset$ during initialization, $\mathcal{U}(\theta) = \emptyset$ before the first execution of DEFINENEW.

Now suppose that $\mathcal{U}(\theta) = \{\theta' \mid \theta \sqsubset \theta' \text{and } \theta' \in \Theta\}$ for all $\theta \in [X \rightharpoonup V_X]$ before an execution of DEFINENEW and show that it also holds after the execution of DEFINENEW. When θ is added to Θ at line 1, θ is also added to $\mathcal{U}(\theta')$ for any θ' with $\theta' \sqsubset \theta$ at line 2. Thus, we still have $\mathcal{U}(\theta) = \{\theta' \mid \theta \sqsubset \theta' \text{and } \theta' \in \Theta\}$ for all $\theta \in [X \rightharpoonup V_X]$. □

LEMMA 2. $\mathcal{U}(\bot) = \Theta$ *before and after each execution of* DEFINENEW.

PROOF. From Lemma 1, we have $\mathcal{U}(\bot) = \{\theta' \mid \bot \sqsubset \theta' \text{and } \theta' \in \Theta\}$. Since $\bot$ is less informative than any parameter instance, $\mathcal{U}(\bot) = \Theta$. □

LEMMA 3. $\Psi \subseteq \Theta$ *before and after each execution of* HANDLEEVENT.

PROOF. A parameter instance θ added to Ψ at line 3 can be added to Θ at line 2. A parameter instance $(\theta_{comp} \sqcup \theta)$ that can be added to Ψ at line 9 is also added to Θ at line 7. □

LEMMA 4. *Before and after each update of* Ψ *in* HANDLEEVENT, *if a parameter instance* $\theta \in \Psi$, *then* θ *is connected.*

PROOF. Before any update of Ψ, the statement holds because $\mathbb{M}\langle X\rangle$ initializes Ψ with $\emptyset$.

Now suppose that the statement holds before an update of Ψ and show that it also holds after the update of Ψ when HANDLEEVENT handles $e\langle\theta\rangle$. There are two places where a parameter instance is added to Ψ: line 3 and line 9 in HANDLEEVENT. We prove that all parameter instances added at line 3 or line 9 are indeed connected.

A parameter instance θ added at line 3 is from the currently handling event $e\langle\theta\rangle$; thus, θ is connected according to Definition 5.

Now, we need to prove that $(\theta_{comp} \sqcup \theta)$ added at line 9 is connected. Since $\theta_i \in \Psi$ from line 8 and the statement holds before this execution of HANDLEEVENT, each θ_i is connected. θ is also connected because HANDLEEVENT is handling $e\langle\theta\rangle$. From the condition at line 6, we have θ_{comp} is compatible with θ. Lastly, the condition at line 8 gives $\theta_i \sqcap \theta \neq \bot$ for each θ_i. Therefore, $\theta_{comp} \sqcup \theta$ is indeed connected. □

LEMMA 5. *For any* $\tau \in \mathcal{E}\langle X\rangle^*$, *after we run algorithm* $\mathbb{M}\langle X\rangle$ *on* τ, *if a parameter instance* $\theta \in \Psi$, *then* θ *is connected.*

PROOF. From Lemma 4, if a parameter instance $\theta \in \Psi$, then θ is connected after an update of Ψ in HANDLEEVENT. This statement holds even after we run $\mathbb{M}\langle X\rangle$, because only HANDLEEVENT updates Ψ. □

LEMMA 6. *For any $\tau \in \mathcal{E}\langle X\rangle^*$, after we run* HANDLEEVENT *for all events in τ, if a parameter instance $\theta \in [X \rightharpoonup V_X]$ is connected, then $\theta \in \Psi$.*

PROOF. $\mathbb{M}\langle X\rangle$ does not have any code that removes an element from Ψ; i.e., Ψ is increasing. Thus, we only need to show that HANDLEEVENT does not miss any connected parameter instance when a new connected parameter instance is created. We can prove this by induction on n when we consider a connected parameter instance as $\theta_1 \sqcup \theta_2 \sqcup ... \sqcup \theta_n$, assuming that $e_i\langle\theta_i\rangle$ occurs before $e_{i+1}\langle\theta_{i+1}\rangle$ for $1 \leq i \leq n-1$.

The statement holds for $n = 1$ because HANDLEEVENT adds a parameter instance θ to Ψ at line 3 when it handles $e\langle\theta\rangle$.

Assume that the statement holds for n and define $\theta_1 \sqcup \theta_2 \sqcup ... \sqcup \theta_n$ as θ_N. It must then be shown that the statement holds for $n+1$; i.e., if $\theta_N \sqcup \theta_{n+1}$ is connected, then $\theta_N \sqcup \theta_{n+1} \in \Psi$. Let us define the new connected parameter instance as $\theta_{N+1} = \theta_N \sqcup \theta_{n+1}$.

When HANDLEEVENT handles $e_{n+1}\langle\theta_{n+1}\rangle$, there are two cases we need to analyze, depending on whether or not there exists j, $1 \leq j \leq n$, such that θ_j is identical to θ_{n+1}. If such θ_j exists, since $\theta \sqcup \theta = \theta$ from Definition **??**, $\theta_1 \sqcup \theta_2 \sqcup ... \sqcup \theta_n \sqcup \theta_{n+1} = \theta_1 \sqcup \theta_2 \sqcup ... \sqcup \theta_n$; therefore, $\theta_N = \theta_{N+1}$. Since this event does not introduce additional connected parameter instance, Ψ still contains all connected parameter instances.

If such θ_j does not exist, we can divide further into two cases, depending on whether or not $\Delta(\theta_{n+1})$ is defined. If $\Delta(\theta_{n+1})$ is undefined, then the branch at lines 2 to 9 will be taken, and the loop at lines 5 to 9 will eventually let θ_{max} be $\perp$ because $\perp \sqsubset \theta_{n+1}$. Then, the loop at lines 6 to 9 will eventually let θ_{comp} be θ_N because $\theta_N \in \Psi$ from the induction hypothesis, $\Psi \subseteq \Theta$ from Lemma 3, and Lemma 2 give $\theta_N \in \mathcal{U}(\perp)$. When θ_{comp} is mapped to θ_N, the condition for this loop is satisfied because $\theta_N \sqcup \theta_{n+1}$ is assumed to be connected and Definition 5 gives θ_N is compatible with θ_{n+1}. Next, θ_N can satisfy the condition at line 8 because the induction hypothesis gives $\theta_N \in \Psi$, and because $\theta_N \sqcup \theta_{n+1}$ is connected and this implies $\theta_N \sqcap \theta_{n+1} \neq \perp$. Thus, $\theta_N \sqcup \theta_{n+1}$ is added to Ψ at line 9.

Now, we need to prove the case when $\Delta(\theta_{n+1})$ is defined. In this case, the connected parameter instance $\theta_N \sqcup \theta_{n+1}$ cannot be added to Ψ because lines from 1 to 9 are skipped; so, we need to prove that $\theta_N \sqcup \theta_{n+1}$ is already in Ψ. If this is proven, Ψ will still contain all connected parameter instances.

Since $\Delta(\theta)$ can be defined only at line 4 in HANDLEEVENT, the fact that $\Delta(\theta_{n+1})$ is defined implies that there existed an event whose parameter binding is θ_{n+1}. Let $e_p\langle\theta_{n+1}\rangle$ be the first event whose parameter binding is θ_{n+1}. It is obvious that θ_{n+1} is connected from Definition 5, and $\theta_{n+1} \in \Psi$.

Without losing any generality, we can define $\mathsf{Dom}(\theta_N)$ and $\mathsf{Dom}(\theta_{n+1})$ as follows:

$$\mathsf{Dom}(\theta_N) = \{x_1, x_2, ..., x_k, y_1, y_2, ..., y_l\}$$
$$\mathsf{Dom}(\theta_{n+1}) = \{x_1, x_2, ..., x_k, z_1, z_2, ..., z_m\}$$

Since it is assumed that $\theta_N \sqcup \theta_{n+1}$ is connected, $\theta_N(x_i) = \theta_{n+1}(x_i)$ for $1 \leq i \leq k$. From Definition **??**, $\mathsf{Dom}(\theta_{N+1}) = \{x_1, x_2, ..., x_k, y_1, y_2, ..., y_l, z_1, z_2, ..., z_m\}$.

Recall that we let θ_N be $\theta_1 \sqcup \theta_2 \sqcup ... \sqcup \theta_n$ and assume $e_i\langle\theta_i\rangle$ occurs before $e_{i+1}\langle\theta_{i+1}\rangle$ for $1 \leq i \leq n-1$. Intuitively, θ_N gradually collects more parameter bindings as more parameter instances are combined. This leads us to define a parameter instance $\theta_{N|i}$, for $1 \leq i \leq n$, such that $\theta_{N|i} = \theta_1 \sqcup \theta_2 \sqcup ... \sqcup \theta_i$. Informally, $\theta_{N|i}$ is the snapshot of θ_N right after θ_i is combined. It is obvious that $\theta_{N|n} = \theta_N$ and each $\theta_{N|i}$ is connected. Also, for any i, $\theta_{N|i} \in \Psi$ because of the induction hypothesis.

Without losing generality, we can consider that $e_p\langle\theta_{n+1}\rangle$ occurs between $e_j\langle\theta_j\rangle$ and $e_{j+1}\langle\theta_{j+1}\rangle$ where $0 \leq j < n$. Note that $e_p\langle\theta_{n+1}\rangle$ cannot be identical to $e_i\langle\theta_i\rangle$, $1 \leq i \leq n$, because this would make $\mathsf{Dom}(\theta_N)$ contain $z_1, z_2, ..., z_m$. When HANDLEEVENT handles $e_p\langle\theta_{n+1}\rangle$, we consider two cases depending on $\{x_1, x_2, ..., x_n\} \cap \mathsf{Dom}(\theta_{N|j}) = \emptyset$.

1) When $\{x_1, x_2, ..., x_n\} \cap \mathsf{Dom}(\theta_{N|j}) \neq \emptyset$, let us pick one element, x_o, from $\mathsf{Dom}(\theta_{N|j})$. When HANDLEEVENT handles $e_p\langle\theta_{n+1}\rangle$, it can choose $(x_o : \theta_{n+1}(x_o))$ as θ_{max} at line 5 because $(x_o : \theta_{n+1}(x_o)) \sqsubset \theta_{n+1}$. It can also choose $\theta_{N|j}$ as θ_{comp} at line 6 because they are compatible as θ_N and θ_{n+1} are compatible. $\theta_{N|j}$ can satisfy the condition at line 8 because $\theta_{N|j} \in \Psi$ from the induction hypothesis, and $\theta_{N|j} \sqcap \theta_{n+1} \neq \perp$ from $\theta_{N|j}(x_o) = \theta_{n+1}(x_o)$. Then, $\theta_{N|j} \sqcup \theta_{n+1}$ is added to Ψ at line 9.

2) $\{x_1, x_2, ..., x_n\} \cap \mathsf{Dom}(\theta_{N|j}) = \emptyset$ implies that all parameter bindings for xs are supplied to θ_N only after $e_p\langle\theta_{n+1}\rangle$ is handled, and there exists $e_q\langle\theta_q\rangle$ such that $j+1 \leq q$ and $\mathsf{Dom}(\theta_q) \cap \{x_1, x_2, ..., x_k\} \neq \emptyset \wedge \mathsf{Dom}(\theta_q) \cap \{y_1, y_2, ..., y_l\} \neq \emptyset$; otherwise, parameter bindings for xs and parameter bindings for ys cannot be connected, and $\mathsf{Dom}(\theta_N)$ cannot have both xs and ys. From $\mathsf{Dom}(\theta_q)$, let us pick one x_o. When HANDLEEVENT handles $e_q\langle\theta_q\rangle$, it can choose $\theta_{N|q-1} \sqcup \theta_{n+1}$ as θ_{comp} at line 6 because $\theta_{N|q-1} \sqcup \theta_{n+1}$ is compatible with θ_q as $\theta_N \sqcup \theta_{n+1}$ is compatible with θ_q. When $\theta_{comp} = \theta_{N|q-1} \sqcup \theta_{n+1}$, line 9 can be reached because both $\theta_{N|q-1}$ and θ_{n+1} can be chosen as θ_i. $\theta_{N|q-1}$ can be selected because $\theta_{N|q-1} \in \Psi$, and because $\theta_{N|q-1} \sqcup \theta_q$ is connected, and thus $\theta_{N|q-1} \sqcap \theta_q \neq \perp$. Also, θ_{n+1} can be selected because $\theta_{n+1} \in \Psi$, and $\theta_{n+1}(x_o) = \theta_q(x_o)$. Then, $\theta_{N|q-1} \sqcup \theta_{n+1} \sqcup \theta_q = \theta_{N|q} \sqcup \theta_{n+1}$ is added to Ψ at line 9.

Both cases show that at some point r, $\theta_{N|r} \sqcup \theta_{n+1}$ is added to Ψ. Now, we prove that for any m, $r \leq m < n$, $(\theta_{N|m} \sqcup \theta_{n+1}) \in \Psi$ by induction on m. For $m = r$, this statement holds as we analyzed for both cases. Assume that the statement holds for m. It must be shown that the statement holds for $m+1$. Since $\theta_{N|m+1}$ is connected, when HANDLEEVENT handles $e_{m+1}\langle\theta_{m+1}\rangle$, there exists θ_{max} that makes it possible to choose $\theta_{N|m}$ as θ_{comp} at line 6, and add to Ψ a parameter instance $\theta_{N|m+1}$ at line 9 by combining $\theta_{N|m}$ and θ_{m+1}. That θ_{max} can also let θ_{comp} be $(\theta_{N|m} \sqcup \theta_{n+1})$ because $\theta_{N|m} \sqsubset (\theta_{N|m} \sqcup \theta_{n+1})$ and $\theta_{N|m} \sqcup \theta_{n+1}$ is also compatible with θ_{m+1}. From the induction hypothesis, $(\theta_{N|m} \sqcup \theta_{n+1}) \in \Psi$. As $\theta_{N|m} \sqcup \theta_{m+1}$ is connected, $\theta_{N|m} \sqcap \theta_{m+1} \neq \perp$ and thus $(\theta_{N|m} \sqcup \theta_{n+1}) \sqcap \theta_{m+1} \neq \perp$; so, $\theta_{N|m} \sqcup \theta_{n+1}$ can be chosen as θ_i, and satisfy the condition at line 8. Therefore, $\theta_{N|m} \sqcup \theta_{n+1} \sqcup \theta_{m+1} = \theta_{N|m+1} \sqcup \theta_{n+1}$ is added to Ψ. From the result of this induction, $\theta_N \sqcup \theta_{n+1} \in \Psi$. □

LEMMA 7. *For any $\tau \in \mathcal{E}\langle X\rangle^*$, after we run algorithm $\mathbb{M}\langle X\rangle$ on τ, if a parameter instance $\theta \in [X \rightharpoonup V_X]$ is connected, then $\theta \in \Psi$.*

PROOF. Lemma 6 shows that if a parameter instance θ is connected, then $\theta \in \Psi$, after we run HANDLEEVENT for all events in τ. Since a connected parameter instance can be created only in HANDLEEVENT, the statement holds. □

LEMMA 8. *For any $\tau \in \mathcal{E}\langle X\rangle^*$, after we run* HANDLEEVENT *for all events in τ, if there is an event $e\langle\theta\rangle$ in τ, $\Delta(\theta)$ is* defined.

PROOF. When HANDLEEVENT handles an event $e\langle\theta\rangle$, it *defines* $\Delta(\theta)$ at line 2. Since a defined $\Delta(\theta)$ never becomes undefined, the statement holds. □

LEMMA 9. *For any $\tau \in \mathcal{E}\langle X\rangle^*$, after we run* HANDLEEVENT *for all events in τ, $\Delta(\theta)$ keeps an event iff its parameter binding is θ.*

PROOF. HANDLEEVENT appends an event to $\Delta(\theta)$ at line 10 when it handles an event $e\langle\theta\rangle$. Thus, $\Delta(\theta)$ keeps an event if its parameter binding is θ. Also, this is the only place where $\Delta(\theta)$ is appended; thus, $\Delta(\theta)$ keeps an event only if its parameter binding is θ. □

Now, we are ready to show that the final result $\Delta(\theta)$ at line 4 in CONSTRUCTCONNECTED is a θ-trace slice $\tau \restriction_\theta$ for the given parametric trace τ. When we prove this in Lemma 10, we will not mention how to preserve the order of events in τ because it is straightforward. When HANDLEEVENT appends an event e at line 10, it also remembers the timestamp that is strictly increasing. The timestamp information allows MERGETRACES to preserve the order when it merges all existing trace slices for θ's sub-instances.

LEMMA 10. *For any $\tau \in \mathcal{E}\langle X\rangle^*$, after we run algorithm $\mathbb{M}\langle X\rangle$ on τ, if θ is connected and $\mathsf{Dom}(\theta) = X$, then $\Delta(\theta) = \tau \restriction_\theta$.*

PROOF. If θ is connected, we have $\theta \in \Psi$ from Lemma 6 because a connected parameter instance can be created only in HANDLEEVENT. Thus, if θ is connected and $\mathsf{Dom}(\theta) = X$, the body of the loop in CONSTRUCTCONNECTED is executed. We only need to prove that $\Delta(\theta)$ returned from MERGETRACES at line 4 is $\tau \restriction_\theta$.

According to Definition 4, $\tau \restriction_\theta$ keeps the base event of $e\langle\theta'\rangle$ iff $\theta' \sqsubseteq \theta$. Showing that $\Delta(\theta)$ returned from MERGETRACES also keeps the base event of $e\langle\theta'\rangle$ iff $\theta' \sqsubseteq \theta$ will complete the proof.

Let us first prove that $\Delta(\theta)$ at line 4 keeps the base event of $e\langle\theta'\rangle$ if $\theta' \sqsubseteq \theta$. Since $e\langle\theta'\rangle \in \tau$, Lemma 8 shows that $\Delta(\theta')$ is defined. As $\Delta(\theta')$ is defined and $\theta' \sqsubseteq \theta$, $\Delta(\theta')$ is added to μ, and MERGETRACES will merge $\Delta(\theta')$. Also, $\Delta(\theta')$ keeps the base event of $e\langle\theta'\rangle$ as Lemma 9 shows. Therefore, $\Delta(\theta)$ contains e.

Next, we need to prove that $\Delta(\theta)$ keeps the base event of $e\langle\theta'\rangle$ only if $\theta' \sqsubseteq \theta$. This statement holds because $\Delta(\theta')$ is considered to be merged at line 3 in CONSTRUCTCONNECTED only when $\theta' \sqsubseteq \theta$, and because $\Delta(\theta')$ keeps an event only if its parameter binding is θ' from Lemma 9. □

Lemma 5, Lemma 7 and Lemma 10 show that Theorem 1 holds.

Proof of Theorem 2

If $\delta(s', e) = s$ exists in A, the loop at lines 8 to 10 in EXPAND introduces $\delta'(s''', e) = s''$ in A' where $s''' \in \gamma(s')$ and $s'' \in \gamma(s)$. EXPAND chooses $i' \in \gamma(i)$ as the initial state of A' at line 12. It also marks all elements of $\gamma(s)$ as final states of A' at line 11 if s is one of the final states in A. Then, for each symbol of $\omega \in T$, if A transitions from s_i to s_j according to δ, A' also transitions from $s'_i \in \gamma(s_i)$ to $s'_j \in \gamma(s_j)$ according to δ'. Thus, if A reaches s_f, then A' reaches $s'_f \in \gamma(s_f)$. If s_f is one of the final states of A, s'_f is one of the final states of A'. Since all transitions needed to accept all strings in T are in δ_r, if A' reaches s'_f using δ', then A' can also reach s'_f using only δ_r, for any $\omega \in T$. Thus, $T \subseteq \mathcal{L}(A')$.

Next, we show that $\mathcal{L}(A') \subseteq \mathcal{L}(A)$. Based on the way EXPAND creates states of A', each state s' in A' has one corresponding state s in A, where $s' \in \gamma(s)$. For this reason, if A' transitions from s'_i to s'_j, then A transitions from s_i to s_j, where $s'_i \in \gamma(s_i)$ and $s'_j \in \gamma(s_j)$. Similarly, the intial state and the final states in A' have corresponding states in A. Thus, if a string is accepted by A', that string is also accepted by A; i.e., $\mathcal{L}(A') \subseteq \mathcal{L}(A)$.